TUTORIAL

How to Build Apps using Redis Streams

Roshan Kumar, Redis

CONTENTS

Redis, the in-memory multi model database, is popular for many use cases. These include supporting content caching, session stores, real-time analytics, message broker and data stream needs. Last year, we published a white paper on how to use Redis for Fast Data Ingest. In that, we described how you could use Redis Pub/Sub, Lists and Sorted Sets for real-time stream processing. Now version 5.0 of Redis has a brand new data structure, Redis Streams, dedicated to managing streams

With the "Redis Streams" data structure, you can do a lot more than what was possible with Pub/Sub, Lists or Sorted Sets. Among many benefits, its features enable you to:

Collect large volumes of data arriving in high velocity (the only bottleneck is your network I/O);

- Create a data channel between many producers and many consumers;
- Effectively manage your consumption of data even when producers and consumers don't operate at the same rate;
- Persist data when your consumers are offline or disconnected;
- Communicate between producers and consumers asynchronously;
- Scale your number of consumers;
- Implement transaction-like data safety when consumers fail in the midst of consuming data and
- Use your main memory efficiently.

The best part of Redis Streams is that it's built into Redis, so there are no extra steps to deploy or manage. In this tutorial, we'll walk you through the basics of how to use Redis Streams, and how consumer groups work, and finally show a working application that uses Redis Streams.

Chapter 1. How to use Redis Streams

The basics of data flow

As described in the tutorial, Redis Streams provides an “append only” data structure that appears similar to logs. It offers commands that allow you to add, consume, get the status and manage how data is consumed. The Streams data structure is flexible, allowing you to connect producers and consumers in several ways.

Figure 1.1. A simple application of Redis Streams with one producer and one consumer.

Figure 1.1 demonstrates basic usage of Redis Streams. A single producer acts as a data source and its consumer is a messaging application that sends data to the relevant recipients.

Figure 1.2. An application with multiple consumers reading data from Redis Streams.

In figure 1.2, a common data stream is consumed by more than one consumer. With Redis Streams, consumers can read and analyze the data at their own pace.

In the next application (shown in figure 1.3), things get a bit more complex. This service receives data from multiple producers, which is all stored in a Redis Streams data structure. The application also has multiple consumers reading the data from Redis Streams, as well as a consumer group, which supports consumers who cannot operate at the same rate as the producers. we will cover this scenario in detail in our next chapter.

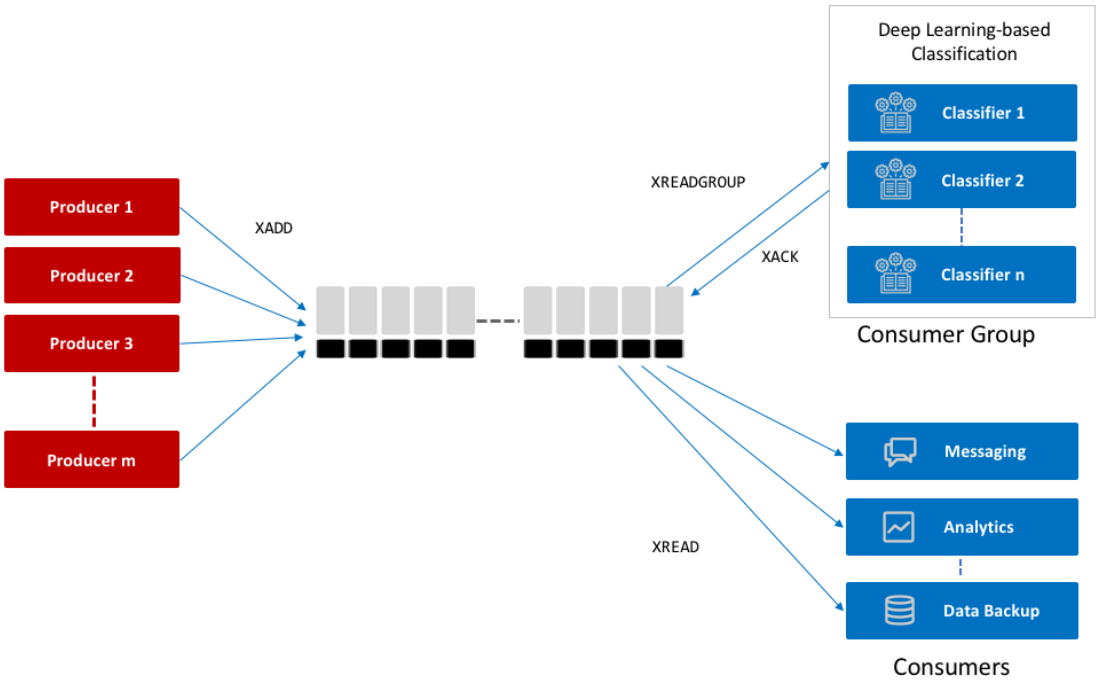

Figure 1.3. Redis Streams supporting multiple producers and consumers.

Adding data to a stream

The diagram in figure 1.3 shows only one way to add data to a Redis Stream. Although one or more producers can add data to the data structure, any new data is always appended to the end of the stream.

1. **The default method for adding data**

This is the simplest way to add data into Redis Streams:

```
XADD mystream * name Anna
XADD mystream * name Bert
XADD mystream * name Cathy
```

In this command, XADD is the Redis command, mystream is the name of the stream, a, b, c are the names added in each line and "*" tells Redis to auto generate the identifier for each line. This command results in three mystream entries:

```
1518951481323-0 name Cathy
1518951480723-0 name Bert
1518951480106-0 name Anna
```

2. **Adding data with user-managed IDs for each entry**

Redis gives you an option to maintain your own identifier for each entry (see below). While this may be useful in some cases, it's usually simpler to rely on auto-generated IDs.

```
XADD mystream 10000000 name Anna
XADD mystream 10000001 name Bert
XADD mystream 10000002 name Cathy
```

This results in the following mystream entries:

```
10000002-0 name Cathy
10000001-0 name Bert
10000000-0 name Anna
```

3. Adding data with a maximum limit

You can cap your stream with a maximum number of entries:

```
XADD mystream MAXLEN 1000000 * name Anna
XADD mystream MAXLEN 1000000 * name Bert
XADD mystream MAXLEN 1000000 * name Cathy
```

This command evicts older entries when the stream reaches a length of around 1,000,000.

Tip: Redis Streams stores data in the macro nodes of a radix tree. Each macro node has a few data items (typically, in the range of a few tens). Adding an approximate MAXLEN value as shown below avoids having to manipulate the macro node for each insertion. If a few tens of numbers - whether 1000000 or 1000050 - makes little difference to you, you could optimize your performance by calling the command with the approximation character(~).

```
XADD mystream MAXLEN ~ 1000000 * name Anna
XADD mystream MAXLEN ~ 1000000 * name Bert
XADD mystream MAXLEN ~ 1000000 * name Cathy
```

Consuming data from the stream

The Redis Streams structure offers a rich set of commands and features to consume your data in a variety of ways.

Read everything from the beginning of the stream

Situation: The stream already has the data you need to process, and you want to process it all from the beginning.

The command you'll use for this is XREAD, which allows you to read all or the first N entries from the beginning of the stream. As a best practice, it's always a good idea to read the data page by page. To read up to 100 entries from the beginning of the stream, the command is:

```
XREAD COUNT 100 STREAMS mystream 0
```

Assuming 1518951481323-0 is the last ID of the item you received in the previous command, you can retrieve the next 100 entries by running:

```
XREAD COUNT 100 STREAMS mystream 1518951481323-1
```

4. Consume data asynchronously (via a blocking call)

Situation: Your consumer consumes and processes data faster than the rate at which it is added to the stream.

There are many use cases where the consumer reads faster than the producers actually add data to your stream. In these scenarios, you want the consumer to wait and be notified with new data as it arrives. Redis Streams' BLOCK option allows you to wait for new data to arrive:

```
XREAD BLOCK 60000 STREAMS mystream 1518951123456-1
```

Here, XREAD returns all the data after 1518951123456-1. If there's no data after that, it will wait for N=60 seconds until fresh data arrives, and then time out. If you want to block this command infinitely, call XREAD as follows:

```
XREAD BLOCK 0 STREAMS mystream 1518951123456-1
```

Note: For this example, you could also retrieve data page by page by using the XRANGE command.

5. Read only new data as it arrives

Situation: You are interested in processing only the new set of data starting from this point in time.

When you are reading data repeatedly, it's always a good idea to restart with where you left off. For example, in the previous example, you made a blocking call to read data greater than 1518951123456-1. However, to start with, you may not know the latest ID. In such cases, you can start reading the stream with the "$" sign, which tells the XREAD command to retrieve only new data. As this call uses the BLOCK option with **60000 milliseconds**, it will wait until there's some data in the stream.

```
XREAD BLOCK 60000 STREAMS mystream $
```

In this case, you'll start reading new data with the $ option. However, you should not make subsequent calls with the $ option. For instance, if 1518951123456-0 is the ID of the data retrieved in previous calls, your next call should be:

```
XREAD BLOCK 60000 STREAMS mystream 1518951123456-1
```

6. Iterate the stream to read past data

Situation: Your data stream already has enough data, and you want to query it to analyze data collected so far.

You could read the data between two entries either in forward or backward directions using XRANGE and XREVRANGE respectively. In this example, the command reads data between 1518951123450-0 and 1518951123460-0:

```
XRANGE mystream 1518951123450-0 1518951123460-0
```

XRANGE also allows you to limit the number of items returned with the help of the "COUNT" option. For example, the following query returns the first ten items between the two intervals. With this option, you could iterate through a stream as you do with the SCAN command:

```
XRANGE mystream 1518951123450-0 1518951123460-0 COUNT 10
```

When you do not know the lower or upper bounds of your query, you can replace the lower bound by "–", and the upper bound by "+". For example, the following query returns the first 10 items from the beginning of your stream:

```
XRANGE mystream – + COUNT 10
```

The syntax for XREVRANGE is similar to XRANGE, except that you reverse the order of your lower and upper bounds. For example, the following query returns the first ten items from the end of your stream in reverse order:

```
XREVRANGE mystream + – COUNT 10
```

7. Partition data among more than one consumers

Situation: Consumers consume your data far slower than producers produce it.

In certain cases, including image processing, deep learning and sentiment analysis, consumers can be very slow when compared to producers. Here, you match the speed of data arriving to the data being consumed by fanning out your consumers and partitioning the data consumed by each one.

With Redis Streams, you can use **consumer groups** to accomplish this. When more than one consumer is part of a group, Redis Streams will ensure that every consumer receives an exclusive set of data.

```
XREADGROUP GROUP mygroup consumer1 COUNT 2 STREAMS mystream >
```

Of course, there's plenty more to learn about how consumer groups work. They are designed to partition data, recover from disasters and deliver transaction data safety. We'll explain all of this in the next chapter.

As you can see, it's easy to get started with Redis Streams -- just download and install Redis 5.0 from redis.io, and get started. In chapter 3, we'll show an example of how to develop an application using the Java-based Lettuce client library for Redis.

Chapter 2. How to use Consumer Groups in Redis Streams

Understanding the data lifecycle in consumer groups

In the last chapter we showed how to add data to a stream, and how you can read the data in multiple ways.

In a perfect world, both data producers and consumers work at the same pace, and there's no data loss or data backlog. Unfortunately, that's not the case in the real world: in nearly all real-time data stream processing use cases, producers and consumer work at different speeds. In addition, there is more than one type of consumer, each with its own requirements and processing pace. Redis Streams appreciates this real need and offers a feature set that gravitates heavily towards supporting the consumers. One of its most important features is a 'consumer group.' In this chapter, you'll learn how a consumer group works.

When should you use a consumer group?

The purpose of consumer groups is to scale out your data consumption process. Let's consider one example – an image processing application. The solution requires three main components:

1. **A producer (one or more cameras, perhaps) that captures and stores images,**
2. **Redis Stream that saves images (in a stream data store) in the order they arrive, and**
3. **An image processor that processes each image.**

Figure 2.1. Sample image processing solution with a single producer and consumer.

Suppose your producer saves 500 images per second, and the image processor can only process 100 images per second at its full capacity. This rate difference will create a backlog, and your image processor won't be able to catch up. An easy way to address this problem is to run five image processors (as shown in figure 2.2), each processing a mutually exclusive set of images. You can achieve this through a consumer group, which enables you to partition your workloads and route them to different consumers.

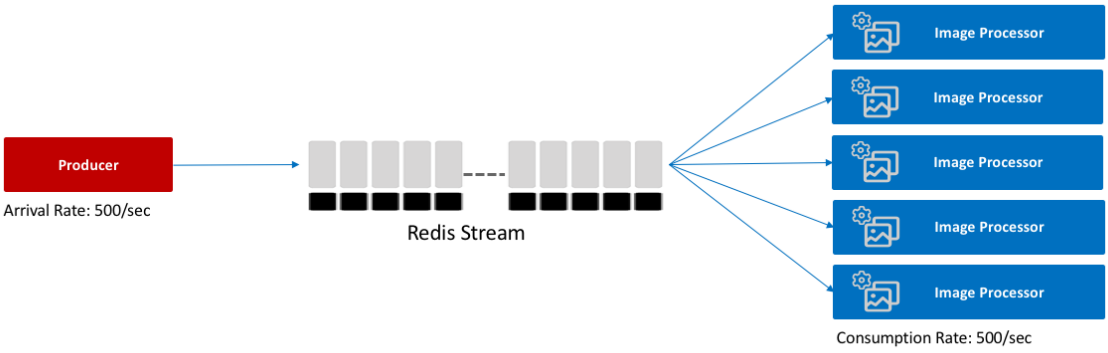

Figure 2.2. Scaling the solution with more consumers to match the production rate.

A consumer group does more than data partitioning – it ensures data safety and enables disaster recovery.

How does a consumer group work?

A consumer group is a data structure within a Redis Stream. As shown in figure 2.3, you can think about a consumer group as a collection of lists. Another thing to imagine is a list of items that are not consumed by any consumers -- for our discussion, let's call this an "unconsumed list." As data arrives in the stream, it is immediately pushed to the unconsumed list.

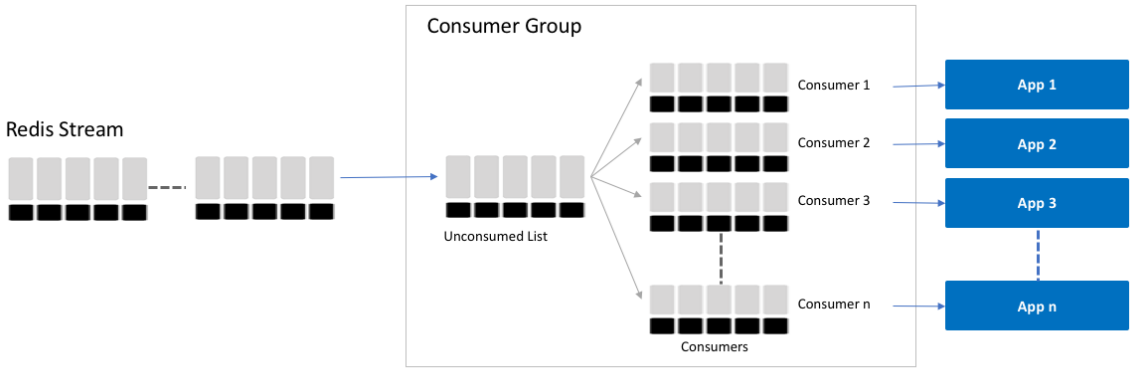

Figure 2.3. How a consumer group is structured.

The consumer group maintains a separate list for each consumer, typically with an application attached. In figure 2.3, our solution has n identical applications (App 1, App 2, ... App n) that read data via Consumer 1, Consumer 2... Consumer n respectively.

When an app reads data using the XGROUPREAD command, specific data entries are removed from the 'unconsumed list' and pushed into the 'pending entries list' that belongs to the respective consumer. Thus, no two consumers will consume the same data.

Finally, when the app notifies the stream with the XACK command, it will remove the item from the consumer's pending entries list.

Now that I've explained the basics of consumer groups, let's dig deeper into how this data lifecycle works.

Creating a consumer group

You can create a new consumer group using the command XGROUP CREATE, as shown below.

```
XGROUP CREATE mystream mygroup $ MKSTREAM
```

As with XREAD, a "$" sign at the end tells the stream to deliver only new data from that point of time forward. The alternate option is 0 or another ID from the stream entry. When using 0, the stream will deliver all data from the beginning of the stream.

MKSTREAM creates a new stream, "mystream" in this case, if it does not already exist.

Reading and managing your data

Assume you have a Redis Stream (“mystream”), and you have already created a consumer group (“mygroup”), as shown above. You can now add items with names a, b, c, d, e as in the following example.

```
XADD mystream * name a
```

Running this command for names a through e will populate Redis Stream, mystream and the “unconsumed list” of the consumer group “mystream”. This is illustrated in figure 2.4.

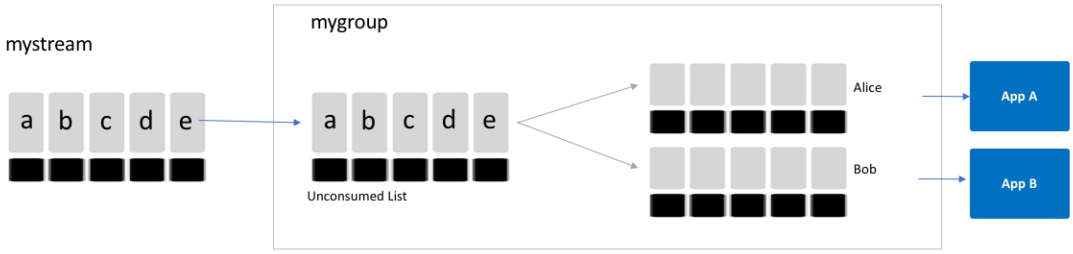

Figure 2.4. A new consumer group, called mygroup.

Using the same names as in Redis Streams Tutorial, here you can see that consumers Alice and Bob haven’t started their jobs yet. App A consumes data through the consumer Alice, while App B does it through Bob.

Consuming data

The command to read data from a group is XREADGROUP. In our example, when App A starts processing data, it calls the consumer (Alice) to fetch data, as in:

```
XREADGROUP GROUP mygroup Alice COUNT 2 STREAMS mystream >
```

Similarly, App B reads the data via Bob, as follows:

```
XREADGROUP GROUP mygroup Bob COUNT 2 STREAMS mystream >
```

The special character “>” at the end tells Redis Stream to fetch only data entries that are not delivered to any other consumers. Also note that no two consumers will consume the same data, which will result in moving data from the unconsumed list to Alice and Bob as shown in figure 2.5.

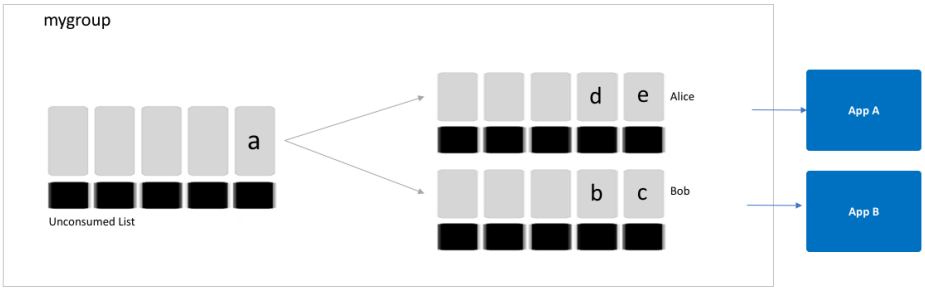

Figure 2.5. Reading data to the consumer in a consumer group.

What happens to data in consumer lists?

The data in your pending entries lists will remain there until App A and App B acknowledge to Redis Stream that they have successfully consumed the data. This is done using the command XACK. For example, App A would acknowledge as follows after consuming “d” and “e” that have the IDs 1526569411111-0 and 1526569411112-0.

```
XACK mystream mygroup 1526569411111-0 1526569411112-0
```

The combination of XREADGROUP and XACK is analogous to starting a transaction and committing it, which ensures data safety.

After running XACK, let’s assume App A executed XREADGROUP as shown below. Now the data structure looks like figure 2.6.

```
XREADGROUP GROUP mygroup Alice COUNT 2 STREAMS mystream >
```

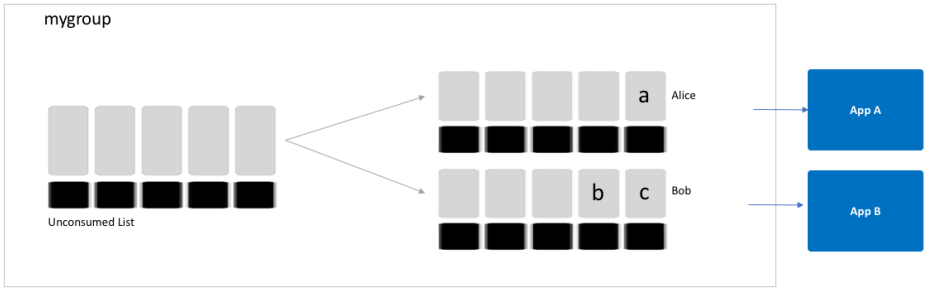

Figure 2.6. The consumer group after XACK and XREADGROUP by App A.

What if App B fails while processing d and c?

If App B terminated due to failure while processing “b” and “c”, then the data structure would look like figure 2.7.

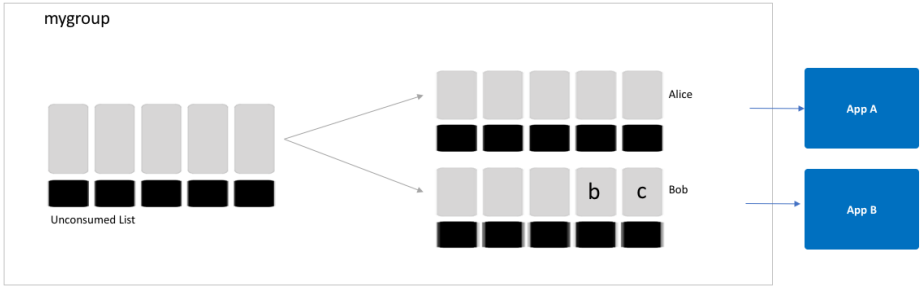

Figure 2.7. App B hasn’t acknowledged with the XACK command.

Now you are left with two options:

1. **Restart App B and reload the data from the consumer (Bob).**

In this case, App B must read data from your consumer (Bob) using the XREADGROUP command, but with one difference. Instead of ">" at the end, App B would pass 0 (or the ID lower than the previous data entry that was processed). If you recollect, ">" fetches new data from the unconsumed list to the consumer.

```
XREADGROUP GROUP mygroup Bob COUNT 2 STREAMS mystream 0
```

The above command will retrieve data entries that are already stored in the list for that consumer. It will not fetch new data from the unconsumed list. App B could iterate through all the data in the consumer Bob before fetching new data.

2. **Force Alice to claim all the data from Bob and process it via App A.**

This is particularly helpful if you cannot recover App B due to node, disk or network failure. In such cases, any other consumer (such as Alice) can claim Bob's data and continue processing that data, thus preventing service downtime. To claim Bob's data, you have to run two sets of commands:

```
XPENDING mystream mygroup - + 10 Bob
```

This will fetch all pending data entries for Bob. The commands '-' and '+' fetch the entire range. If "b" and "c" had the IDs 1526569411113-0 and 1526569411114-0 respectively, the command that will move Bob's data to Alice is below.

```
XCLAIM mystream mygroup Alice 0 1526569411113-0 1526569411114-0
```

Consumer groups maintain a running clock for data in the consumed list. For example, when App B reads "b", the clock kicks in until Bob receives the ACK. With the time option in the XCLAIM command, you can tell the consumer group to move only data that's idle longer than a specified time. You can also ignore that by passing 0 as shown in the example above. The result of these commands is illustrated in figure 2.8. XCLAIM also comes in handy when one of your consumer processors is slow, which results in a backlog of unprocessed data.

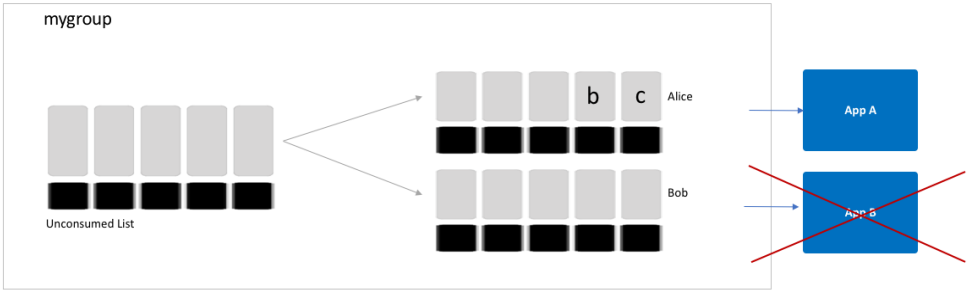

Figure 2.8. Alice claimed all the data from Bob.

We're not done, yet. There's more to come.

In our previous chapter, we covered the basics of how to use Redis Streams. We went a bit deeper in this chapter and explained when to use consumer groups and how they work. Consumer groups in Redis Streams reduce your burden when it comes to managing data partitions, their lifecycles and data safety, and the scale-out benefits of consumer groups helps power many real-time applications. In our last chapter of this tutorial, we will demonstrate how to develop a real-time classification application using Redis Streams and Lettuce, a Java-based open source library for Redis.

Chapter 3. How to build a Redis Streams application

Exploring a sample end-to-end solution using Redis Streams

In the first chapter, we covered the basics and benefits of Redis Streams. In the previous chapter, we described how you can scale out data stream consumers using consumer groups. In this final chapter, we'll demonstrate how to develop applications using Redis Streams.

Cataloging influencers on Twitter

In order to bring Redis Streams to life, let's build a real-world solution, which we'll call "TopSocialForce." This application will gather social messages from Twitter, store them in Redis Streams, and analyze them in real time to identify top influencers. TopSocialForce will do this by tagging any Twitter handle with more than 10,000 followers as an "influencer" and maintain a running catalog.

TopSocialForce collects Twitter data stream through a third party service, like GNIP or PubNub. The tweets follow the JSON format as described by Twitter, so every tweet contains the Twitter handle and its follower count, which we'll use for influencer classification.

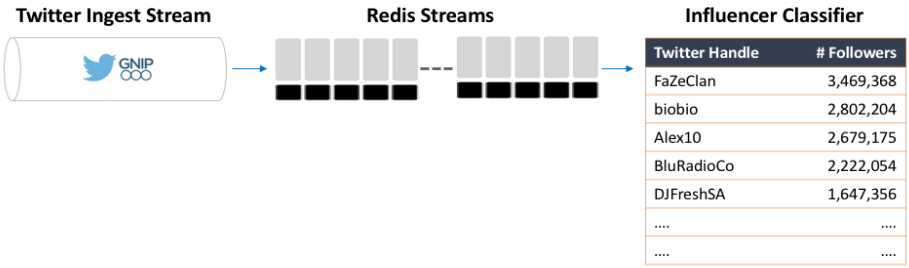

Figure 3.1. How TopSocialForce works

Data Streams processing poses a variety of challenges

With TopSocialForce, we have a few challenges to address. The tweets arrive at a random pace and at the rate of thousands per second. When an influencer tweets a message, we have to leverage that opportunity to tag the influencer, as we cannot predict when that account will tweet again. Therefore, we cannot afford to skip any tweet. TopSocialForce must not only be resilient to connection loss or software failures, but also have the appropriate database and data structures to save and process all tweets.

The ideal data stack for streams

A Redis database with the Redis Streams data structure is the perfect solution to collect, store and drive the processing of Twitter feeds. In this example, we'll show how we developed TopSocialForce on the Java platform with a Redis client library called Lettuce. Redis client libraries are available in other programming languages too, so you can select whichever language platform and library best suits your particular requirements.

Since TopSocialForce receives the Twitter stream from a service called PubNub, our stack also imports the Java library for PubNub.

Specifically, the components of our stack include:

1. **Database: Redis 5.0 (https://redis.io).** The new Redis Streams data structure is available in version 5.0.

2. **Language Platform: Java 1.** We selected Java for the sake of its popularity, but you could write your applications in many other programming languages that support Redis Streams. Check the full list of clients for more information.

3. **Sample Twitter Feed: PubNub.** PubNub offers a streaming API for Twitter data, with a free version that delivers a limited set of tweets for developers. Since we developed TopSocialForce as a proof of concept, we used PubNub's free service.

4. **Redis Library: Lettuce 5.0.** Lettuce is a popular Java client library for Redis and supports Redis Streams starting with version 5.0. At the time of writing, Lettuce 5.0 was a beta release. (Note: Jedis and Redisson are the other Java libraries that support Redis Streams.)

Figure 3.2. The solution stack

The Solution: Twitter ingest stream + Twitter influencer classifier

TopSocialForce consists of two primary processes: a Twitter ingest stream and a Twitter influencer classifier. Both processes use Redis Stream as the underlying stream database to store the Twitter feed.

Ensuring the solution is resilient to connection loss

Traditional messaging modules such as publish/subscribe require that publishers and subscribers are in execution and connected to a common channel all the time. However, Redis Streams does not have that limitation. In contrast, Redis Streams provides a persistent data store for the streaming data. As shown in figures 3.3 and 3.4 below, our Redis Stream decouples the ingest stream from the influencer classifier.

Figure 3.3. The data ingest stream continues collecting data even when consumers are disconnected.

Figure 3.4. The consumer must wait for new data to arrive when the ingest stream is disconnected.

Ensuring data won't be lost in transit

Redis supports both persistence and replication to ensure your data will not be lost. With persistence enabled, Redis saves data to disk in case it needs to be recovered. With replication, you can also deploy a replica server that stores a copy of the data so the data at rest is durable. However, when a consumer consumes the data, typically it moves out of Redis to the consumer's process. If the data is lost during that transition, it may get lost forever. Redis Stream mitigates this problem with the help of explicit 'ack' commands in consumer groups. We covered consumer groups in detail in the previous chapter.

In our solution, the influencer classifier process has a consumer that belongs to the consumer group, called "InfluencerConsumerGroup." The lifecycle of the data in transit is as follows:

1. Our influencer classifier calls the XGROUPREAD command of the Redis Stream.
2. In response to the call, Redis does two things.
   a. Copies the latest data object to the pending list of objects for that consumer and
   b. Delivers the object to the consumer.
3. The consumer (influencer classifier) processes the data and sends XACK. This causes Redis to remove the data from the pending list.

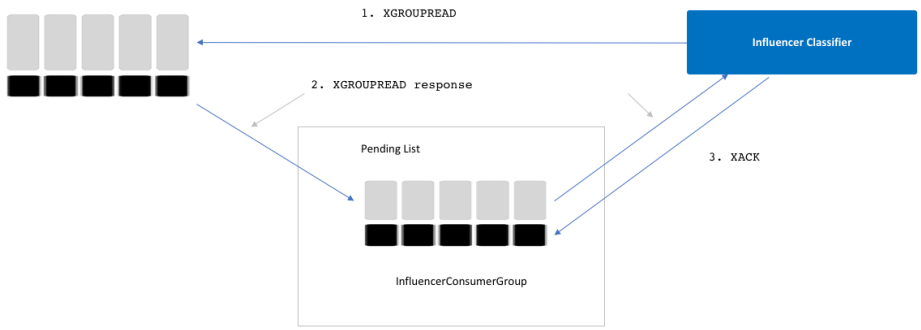

Figure 3.5. Data lifecycle ensures data safety during transit.

If the consumer loses the data in transit or while processing, it still remains in the pending list, and hence is not lost. The influencer classifier process can always retrieve data from the pending list using the XGROUPREAD command.

Influencer catalog data structures

TopSocialForce uses the built-in data structures of Redis – Sorted Set and Hash – to store our influencer data. In the Sorted Set, we store the Twitter handles and their follower counts as each score (see figure 3.6). We use the Hash data structure with a key for each influencer to store their account details. The Hash is indexed by key, influencer:[twitter handle]. Figure 3.7 shows a sample Hash data structure.

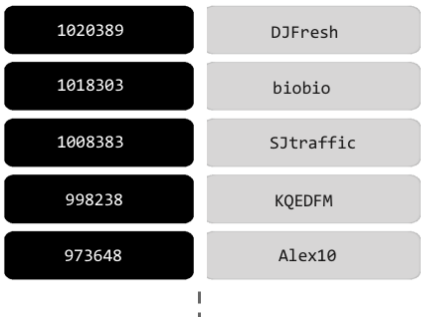

Figure 3.6. The Sorted Set data structure stores the influencers.

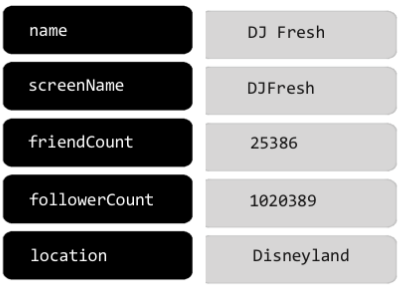

Figure 3.7. The Hash data structure stores influencer details.

Program Design: Class hierarchies and relationships

Since we used Java as our language platform, we leveraged Java's object-oriented programming features to make TopSocialForce easily extensible. The UML class diagram in figure 3.8 gives a general idea of our class structure, and you can find this complete package on github. In the following section, we'll explain at a high level how the programs are organized and what they do. Feel free to read the documentation inside the programs themselves for further details.

1. **Common components**
    a. **LettuceConnection.java:** This is a utility class that manages our connection to the Redis database via the Lettuce library. Other classes in the package use LettuceConnection objects to connect to Redis.
    b. **InitializeConsumerGroup.java:** In this class, we created the Redis Stream data structure and a new consumer group. By doing so, we initialized the database to receive new data from the producer and pass it on to the consumers. This program only needs to be run once to initialize the data structure.

2. **Ingest stream** (the producer component of TopSocialForce)
    a. **IngestStream.java:** This is a parent class with the generic set of properties and methods required for any kind of ingest stream program.
    b. **TwitterIngestStream.java:** This class 'extends' IngestStream, and therefore inherits all its protected and public properties and methods. In addition to the generic features, TwitterIngestStream adds features that are specific to Twitter's data stream.

3. **Influencer classifier** (the consumer side of TopSocialForce)
    a. **InfluencerCollectorMain.java:** This is the 'main' Java class that starts the process.
    b. **MessageProcessor.java:** TMessageProcessor is a Java interface that defines the processMessage() method.
    c. **StreamConsumer.java:** This is a generic class that reads new data as it arrives in our Redis Stream. StreamConsumer extends Thread, and therefore runs as a separate thread. A StreamConsumer object passes on new data to the registered MessageProcessor object.
    d. **InfluencerMessageProcessor.java**: This class implements a MessageProcessor interface. In our program, we pass an InfluencerMessageProcessor object as the type MessageProcessor to StreamConsumer. InfluencerMessageProcessor then stores our influencer data back in Redis using Sorted Set and Hash data structures. You can implement your own version of MessageProcessor and pass it onto StreamConsumer.

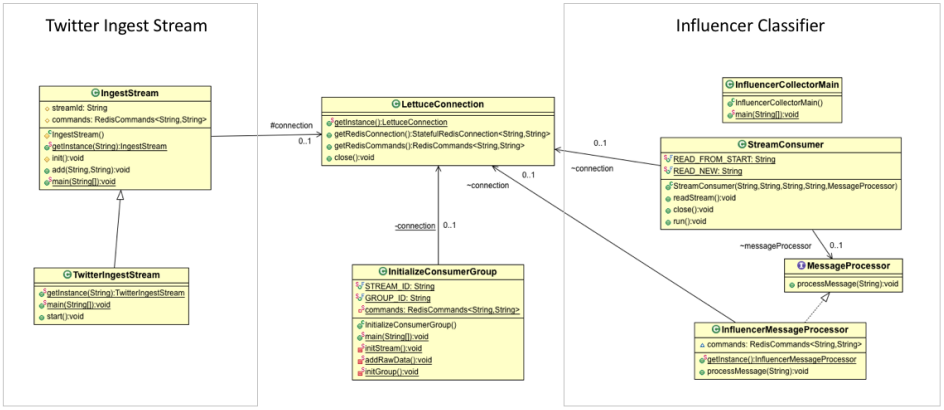

Figure 3.8. The Java classes as a UML class diagram.

Running and testing the program

Our documentation on github provides details on how to compile and run these programs. However, before you get started, make sure you have the right connection parameters set in LettuceConnection.java. In a nutshell, just follow these steps:

1. **InitializeConsumerGroup:** This program initializes a Redis Stream and its consumer groups. You must run this before you collect and process Twitter data, and it will persist after initializing the data structures.

2. **TwitterIngestStream:** Before you compile TwitterIngestStream.java, make sure you have updated your PubNub connection credentials. When you compile and run this program, it connects to PubNub and starts receiving the Twitter data stream. The program then stores the data using the Redis Streams data structure we initialized in the previous program. It reads the Twitter data continuously and does not terminate on its own.

3. **InfluencerCollectorMain:** This class starts the StreamConsumer as a separate thread. StreamConsumer reads the new data in your Redis Stream and passes it onto the MessageProcessor object. StreamConsumer's thread makes a blocking call to Redis Streams. Therefore, it does not terminate on its own, but waits for new data in the Redis Stream if there's none.

When you execute TwitterIngestStream and InfluencerCollectorMain, make sure to test for the following:

1. Both TwitterIngestStream and InfluencerCollector are running
2. More instances of InfluencerCollector objects consuming the data
3. Failure scenario: TwitterIngestStream is down
4. Failure scenario: InfluencerCollector is down

Verifying the data

There's more than one way to verify your data. Here we show a few sample commands using the redis-cli interface:

1. **Data in Redis Streams:**
   a. **Is the data growing**:

```
XLEN twitterstream
```

Run this command a few times to see if the count is changing.

   b. **How can I know more about the data in Redis Streams**?

```
XINFO STREAM twitterstream
```

This command prints basic information such as the stream's length, last generated id, first entry, last entry, etc.

   c. **Can I get a list of the consumer groups attached to my stream**?

```
XINFO GROUPS twitterstream
```

This command lists all the groups associated with the stream.

d. **Are the consumers of a consumer group idle or running**?

```
XINFO CONSUMERS twitterstream influencerclassifiers
```

This lists all the consumers with the consumer group influencerclassifiers, their idle time and the count of pending items to process.

2. **Influencers catalog**
   a. **How many influencers have I collected so far?**

```
ZCARD infuencers
```

   b. **What about the details about an influencer, say ABCD**?

```
XINFO STREAM twitterstream
HGETALL influencers:ABCD
```

This is just the beginning

In this tutorial, we started with an introduction to how you could use Redis Streams for different use cases, and in the second chapter, we went through the details of your data lifecycle, both from the producer and consumer point of views. In this final chapter, we demonstrated how you could apply the concepts we covered earlier to develop an end-to-end application using Redis Streams.

As you scale up an application like this, you'll really appreciate the beauty of Redis Streams because it remains sleek, simple and solid no matter how many data producers or consumers you're managing. The intention of this article series was to seed some ideas for how you could start to benefit from the new Redis Streams data structure, which is available in Redis starting from version 5.0. Redis Streams is also available in Redis Enterprise, which offers high availability and durability with in-memory replication.

Now that you have this foundation, don't hesitate to get started with Redis Streams. It's just the beginning of endless possibilities.

700 E El Camino Real, Suite 250
Mountain View, CA 94040
(415) 930-9666
redis.com